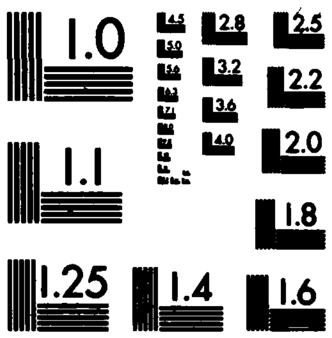
1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

Functional Semantics

Richard Hamlet
and
Harlan Mills

Department of Computer Science
University of Maryland
College Park 20742

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

- a -

Functional Semantics

Richard Hamlet
and
Harlan Mills

Department of Computer Science
University of Maryland
College Park 20742

Abstract

A Analysis of computer programs using a semantics that combines features of the operational and denotational methods is described. *The method is an explanatory, analytic tool, a program calculus* that allows program meaning to be obtained from program syntax, and compared to a desired meaning. Meanings are functional, sets of ordered (input, output) pairs. A subset of Pascal is used to illustrate the theory.

Acknowledgements

Dr. Hamlet's research was supported by the Air Force Office of Scientific Research (Contract F49620-80-C-0001). Computer support was provided by the University of Maryland Computer Science Center and the Department of Computer Science.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-0305	2. GOVT ACCESSION NO. AD-A127372	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FUNCTIONAL SEMANTICS		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Hamlet and Harlan Mills		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE 1982
		13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Analysis of computer programs using a semantics that combines features of the operational and denotational methods is described. The method is an explanatory, analytic tool, a "program calculus" that allows program meaning to be obtained from program syntax, and compared to a desired meaning. Meanings are functional, sets of ordered (input, output) pairs. A subset of Pascal is used to illustrate the theory.

DD FORM 1 JAN 73 1473

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

83 04 27 020

Introduction

Three important theoretical underpinnings of computer programming are:

(1) Syntax. Phrase-structure grammars (usually context-free) are used to specify the form programs may take [1]. It is possible to give very precise descriptions using more complex grammar mechanisms [2,3], but a collection of "static semantics" informally given is the rule [4]. In any case, there is almost no disagreement about the appropriateness of defining language syntax using grammars.

(2) Semantics. Meanings can be assigned to programs in ways ranging from clever use of natural language [1] through semi-formal aids [5] to careful mathematical definitions [3,6,7]. Syntax is exploited to break a program into units whose separate meanings are defined, then combined to form the meaning of the whole. However, there is no consensus about the "best" semantic definitional technique. Furthermore, it is easy to confuse definition with specification (see (3) following) because each technique carries with it a natural method of reasoning about the meanings it defines.

(3) Specification. The definition of programming-language semantics captures what a program *does* mean; specification captures what one *was intended* to mean. It is natural to use a technique for specification that is closely related to the definitional technique--then it is possible to *reason* about programs, that is, *prove* things about them and their specifications.

This paper describes the so-called "functional semantics" of Mills [8,9], which combines features of the operational and denotational approaches. Care is taken to separate definitions of semantics from specifications, and reasoning about programs from both. The technique is both intellectually satisfying and practical.

It may be helpful to describe our goals in terms of the best-known alternative semantic/specification theory, the Floyd/Hoare assertion technique. (In the discussion to follow, the ideas of "truth" and "proof" will be treated informally, to make clear the computer-science issues at the expense of the logical ones. For a complementary treatment, see Apt [10].) If the paper were explaining the Floyd/Hoare technique it would first define the meaning of the syntactic building blocks of programs (expressions, assignments, conditionals, loops, procedures, etc.). Since the Floyd/Hoare method uses assertions in a first-order theory whose syntax overlaps that of the program, these meanings take the form of verification conditions, statements built for a language construction from assertions attached before and after it. The verification condition for (say) assignment V_A is thus a *definition* of meaning: given pre-assertion P and post-assertion Q, $V_A(P,Q)$ holds just in case the truth of P followed by execution of the assignment guarantees the truth of Q.

Specifications for the Floyd/Hoare method consist of a pair of assertions, the first (input) constraining initial values and the second (output) prescribing final values. That is, the desired behavior is that when any values satisfying the input assertion are provided, a result is to be computed that will satisfy the output assertion. (In the the so-called "partial correctness" form, the result need only satisfy the assertion if it is forthcoming. This is obviously a poor sort of specification, since it can be universally met by a nonhalting program.)

Finally, with both semantics and specification defined, it is possible to reason about programs. At its most abstract, this reasoning takes the form of trying to devise assertions that separate each program construction whose meaning has been isolated. With given input and output assertions, this determines all the verification conditions, and proving that each is true establishes that the program meets its specification. The practical application of the Floyd/Hoare method then reduces to finding appropriate "intermediate assertions" to go with a program, ones which lead to verification conditions that can be established. In many cases there are natural choices for these assertions, choices which *guarantee* that the verification conditions are true. For example, if the assertion before an assignment is the same as that after it, but with the assigned expression substituted for the variable assigned to, then the defining verification condition holds and need not be considered each time.

For complex constructions of a programming language, the necessary intermediate assertions can best be determined by examining the program and trying to capture in an assertion just what is in fact true at the intermediate point. A substantial component in establishing verification conditions involves not their form arising from the language definition, but their inclusion of the pre- and post-assertions. Since those assertions are concerned with what is true at a point in a program, they express facts about program values and operations. In this way much of a program's proof comes to be concerned with its subject matter. For example, number theory will be used to prove programs involving counting; the theory of permutations will be used in sorting, and so on.

In summary, were we treating the Floyd/Hoare method we would:

(1) Define each programming language construction by giving its verification condition. (The "constructions" are those based on a syntactic decomposition into appropriate units sufficient to construct all programs.)

(2) Define specification by a pair of input/output assertions, and define correctness of a program relative to a specification as: if the input assertion is true, the program must execute so as to make the output assertion true.

(3) Describe practical methods of discovering intermediate assertions so that the task of establishing that a program meets its specifications is simplified and mechanized insofar as possible.

(In passing, we note that such a treatment of Floyd/Hoare logic is not easily available. Many of its supporters could supply it, and it could be obtained from the literature with only routine (but substantial) work. Nevertheless, students of computer science often learn the method without much understanding.)

There is a further component to a semantic theory that we do not intend to address here, which for the Floyd/Hoare method would be as follows:

(X) Give rules for design and construction of programs to given specifications, such that the program and its assertions (and proof) fit together naturally.

This final step is the most difficult, but the most important in the application of a semantic theory. However, it is inessential for understanding. It is quite possible to have a satisfying, revealing explanatory theory about the world which is yet difficult to apply in practical cases. In this paper we stress analytic, explanatory ideas, and avoid the ideas of synthesis. It is not that our theory is deficient in this area. To the contrary, it is used as a practical design method in industry; this paper deals with its theoretical underpinnings.

The rest of this paper is organized as follows:

2 The Language

In choosing a programming language to explain, when the subject is not that language but the explanation, there is a fine line between the real and the abstract. On the side of abstraction, it can be argued that the language should show off the features of the explanation to good advantage, and the peculiarities of real languages should not be allowed to confuse a clear picture. But the real-side argument is equally cogent: if the explanation cannot easily handle complications that exist in practice, it is a failure.

As primitive data types we include only character data and files of characters. The virtue in this choice over numeric types is that it eliminates number theory--not the central language-theory idea--from the example. There are few natural operations on characters, but comparisons, and the operation of "next in alphabetic sequence" are included; the latter is undefined at the end of the alphabet.

Statements of our language present less difficult choices. There is an assignment, read/write, a conditional, iteration, and procedure invocation. The most important decision here is to allow the creation of intermediate results through assignment (and thus deal with "sequential" instead of "functional" programming), and to make statement sequence the fundamental construction of the language.

Declarations must be treated by any reasonable semantic theory, but block structure is not central to our language. The idea of two identical identifiers with different meanings is retained by allowing declared procedures to have local variables.

Of the many combinations of procedure call and parameter mechanisms, we have chosen call-by-reference and recursive procedures as the ones to include.

The language that results from these choices is probably closer to IAL (of which MAD [11] and JOVIAL [12] are common examples) than to Pascal, but since the latter is much better known, we express our programs in a subset of Pascal called "CF Pascal" (for Character, File). Although for present purposes it is irrelevant that this language is of practical use, we have found it so for text-formatting problems, and for teaching introductory programming [13]. The exact scope of the CF subset is best defined by the semantics to follow: CF Pascal includes what we define, and where the Pascal definition [14] is clear, we define it that way.

3 Functions and Data States

CF Pascal programs, viewed as "black box" objects, have two special files, INPUT and OUTPUT; a program transforms the former into the latter. Because these files contain character sequences, the meaning of a program is a string-to-string mapping. It is the "denotational" view of semantics (usually credited to Scott [15]) to assign to a program text (itself of course a string) a meaning from the string-to-string maps, and to construct this meaning by first assigning appropriate mappings to program parts, then combining them into a meaning for the entire program. To the contrary, in the "operational" view (which goes back at least to Turing [16]) of semantics, the central role is played by an internal program state, and the parts of the program are viewed as authorizing transformations of that state. The program's function is then the collection of all pairs that begin and end a transformation sequence called a "computation."

The semantic theory to be presented here uses both ideas. We make essential use of an internal state and its transformation, but we express the state-to-state maps denotationally instead of speaking about computation sequences. The result combines the low-level, step-by-step intuition of the Turing approach with the clarity of Scott's overall, functional view.

3.1 The CF Pascal Data State

With only characters and files of characters, the values associated with quantities internal to CF Pascal programs come from an alphabet and strings over that alphabet, with the complication that files are "marked" to separate the part already processed from that yet to be processed. That is, a file is technically a pair consisting of a "past" string and a "future" string. Thus a *data state* of a CF Pascal program includes values from the two sets: characters and string pairs. Each value is connected with a program identifier, and represents its current "contents." In the denotational view a data state is therefore a mapping from identifiers to values. For example, in a program where the identifier OUTPUT (of type TEXT) and XXX (of type CHAR) occur (note the special type font employed for parts of programs), a data state T might be:

T = {(OUTPUT,(Page 1, Λ)), (XXX,2)}

where strings such as Page 1 are shown in a special type font, and Λ is the empty string. However, when no confusion can arise, we will display data-state examples in a simpler notation, using the type fonts to distinguish values from identifiers. For example, the data state T above will be written:

T = (OUTPUT•Page 1_, XXX•2)

where the "*" separates identifier and value, and an underline marks the first character of the future string in a file. In the example, OUTPUT's future string is empty. As an example of the denotational view, in which *T* is a mapping, we would write in this case:

T(XXX) = 2 .

To express the transformation of one data state to another, our notational device is to employ the program fragment that effects the transformation, but to distinguish that fragment from a string (the program syntax) by surrounding it with a box. (The idea is due to Kleene [17].) Thus the denotational view that the meaning of a fragment is a mapping, and meaning itself the association of a program string with its data-state map, is expressed by boxing the string and being explicit about the state. For example, in the state *T* above, the transformation effected by

```
WRITE('3')
```

would be shown as

$\boxed{\texttt{WRITE('3')}}\,(T)$ = (OUTPUT*Page 13_, XXX*2).

The notation is excellent for expressing examples; when we try to give the general case (for example if *T* were not a particular data state, but *any* state) it works less well. The functional notation using ordered pairs is an improvement. For this example,

$\boxed{\texttt{WRITE('3')}}$ = {(*T*, *U*): *T* = *U* except that the past string of
U(OUTPUT) is the past string of
T(OUTPUT) with 3 appended}.

3.2 Meaning of Expressions

The meanings of CF Pascal statements and programs are constructed from the meanings of CF Pascal expressions. Because the language is severely restricted, there are very few of these. The meaning of an expression is a mapping from data states to the appropriate value range (characters for CHAR expressions, string pairs for files, and {true, false} for Boolean expressions). If *X* is a variable (necessarily of type CHAR or TEXT), then

$\boxed{X}\,(T) = T(X)$.

For example, if

T = (OUTPUT*Page 1_, XXX*2)

then

$\boxed{\texttt{OUTPUT}}\,(T)$ = Page 1
$\boxed{\texttt{XXX}}\,(T)$ = 2.

(The file notation does not distinguish a blank future string from an empty one; rather than introduce a visible "blank" character, we will avoid blanks in examples where they would cause trouble.)

The constants in CF Pascal are of type CHAR, and their meanings are the obvious ones:

$\boxed{\texttt{'A'}}(T) = \mathbb{A}$
$\boxed{\texttt{'B'}}(T) = \mathbb{B}$
etc.

for all data states T.

The only other CHAR expression uses the built-in function SUCC, and its meaning is defined inductively. As a first attempt, we might try:

$\boxed{\texttt{SUCC(}E\texttt{)}}(T)$ = the character following $\boxed{E}(T)$ in sequence,

where E is any CHAR expression, and "in sequence" is the lexicographic character ordering defined for Pascal. Because the character ordering may differ from machine to machine, and to illustrate the treatment of runtime errors, we here take SUCC to be defined on $\mathbb{A}$ through $\mathbb{Y}$ in the obvious way, but make

$\boxed{\texttt{SUCC('Z')}}$

undefined for all data states. This statement could be added to the English of the trial definition above, but instead we move the data state into the defining condition, and give the meaning function itself as a collection of ordered pairs:

$\boxed{\texttt{SUCC(}E\texttt{)}}$ = $\{(T, c)$: $\boxed{E}(T)$ is the character b, and c follows b in sequence$\}$.

In this definition, $\boxed{\texttt{SUCC(}E\texttt{)}}$ can fail to be defined at a particular input state T for two reasons. It may happen that $\boxed{E}$ is not defined at T (and hence $\boxed{E}(T)$ is no character b as required), or that b has no following character c. In either case, this T never appears paired with any c in the defining set.

Finally, Boolean expressions can be given meaning. Only single comparisons between character expressions are part of CF Pascal, and the definitions all have the form:

$\boxed{E < F}(T)$ = $\underline{\text{true}}$ if $\boxed{E}(T)$ precedes $\boxed{F}(T)$; $\underline{\text{false}}$ otherwise

for all expressions E and F (and similarly for the operators other than <). In contrast to the treatment of SUCC, we here assume that each pair of characters can be meaningfully compared, but only obvious situations like $\mathbb{B} < \mathbb{X}$ will occur in examples. Care is required here because of the occurrence of the box functions on the right side of the definition. Should one of them be undefined, then the meaning function of the Boolean expression is also undefined. This corresponds to the Pascal convention of complete evaluation (as opposed to McCarthy evaluation) of Boolean expressions, and to the arbitrary choice that once a runtime error occurs, the failure of meaning propagates. Failure to be careful about such definitional matters has long been a source of trouble in program proving [18].

The definition of meaning for CF Pascal expressions is now complete.

3.3 External Strings and Data States

The meaning of a program must be a string-to-string function; the meaning of a program fragment is a data-state-to-data-state function. To bring these into line requires associating a string that is presumed to constitute the input file with an appropriate internal data-state string pair attached to INPUT, and similarly associating the internal OUTPUT pair (initially empty in both past and future parts) with the program output. The necessary actions can be imagined to be the meaning functions of the program header and terminating period, program parts that therefore transform strings to states, and states to strings, respectively.

$$\boxed{\texttt{PROGRAM } P\texttt{ (INPUT, OUTPUT)}}\,(D)$$
$$= (\texttt{INPUT*}\underline{D}, \texttt{OUTPUT*}\_)$$

$$\boxed{\texttt{.}}\,(\ \ldots, \texttt{OUTPUT*}D\_, \ldots) = D$$

where $\underline{D}$ means the empty string Λ paired with D: (Λ, D), that is, D with its first character marked; and $D\_$ means the string pair (D, Λ), that is, D with the mark on an empty string following its last character.

3.4 Calculating Program Meaning

The semantics to be presented below is a functional "calculus" that *allows step-by-step* computation of a program's meaning. By anticipating some of the definitions to be subsequently presented, we can now illustrate this calculus. For the purposes of illustration, the following are special cases which will appear later in more general form:

$$\boxed{\texttt{BEGIN END}} = I$$

$$\boxed{\texttt{BEGIN IF } E \texttt{ THEN END}} = \{(T, T): \boxed{E} \text{ is defined at } T\}$$

where I is the identity function. (In the second case the function is either I or a subfunction of I defined just where $\boxed{E}$ is defined.) A program's meaning is defined to be the (functional) composition of the meanings of its fragments, taken in order. Then we can calculate:

$$\boxed{\texttt{PROGRAM P1(INPUT, OUTPUT); BEGIN END.}}\,(D)$$
$$= \boxed{\texttt{BEGIN END.}}\,(\boxed{\texttt{PROGRAM P1(INPUT, OUTPUT)}}\,(D))$$
(meaning of the program is the composition of its parts' meanings)
$$= \boxed{\texttt{BEGIN END.}}\,((\texttt{INPUT*}\underline{D}, \texttt{OUTPUT*}\_))$$
(definition of the meaning of the header)
$$= \boxed{\texttt{.}}\,(\boxed{\texttt{BEGIN END}}\,((\texttt{INPUT*}\underline{D}, \texttt{OUTPUT*}\_)))$$
(composition of parts again)
$$= \boxed{\texttt{.}}\,(I((\texttt{INPUT*}\underline{D}, \texttt{OUTPUT*}\_)))$$
(the meaning of the null statement is the identity function I)
$$= \boxed{\texttt{.}}\,((\texttt{INPUT*}\underline{D}, \texttt{OUTPUT*}\_))$$
(by definition of I)
$$= \Lambda$$
(definition of the meaning of the final period).

And also,

$$\boxed{\texttt{PROGRAM P2(INPUT, OUTPUT); BEGIN IF SUCC('Z') < 'A' THEN END.}}\,(D)$$

$= \boxed{.}\,(\boxed{\texttt{BEGIN ... END}}\,((\texttt{INPUT*}\underline{D}, \texttt{OUTPUT*}\_)))$
(as above).

The function

$\boxed{\texttt{BEGIN IF SUCC('Z') < 'A' THEN END}}$

is by definition either the identity function or undefined, depending on the evaluation of the two CHAR expressions. Both have (or fail to have) values independent of the state, namely

$\boxed{\texttt{SUCC('Z')}}\,(T)$ = character following $\boxed{\texttt{'Z'}}\,(T)$ (if any)
= character following $\mathbb{Z}$ (if any)

but in fact there is none. It is therefore irrelevant that

$\boxed{\texttt{'A'}}\,(T) = \mathbb{A}$;

the inner-most function is undefined, and the result is that the program P2 means a function that is everywhere undefined, that is, the empty function.

4 Linear Programs

This section includes the definitions of meaning for each imperative statement of CF Pascal, by subsection:

<u>Subsection</u>	<u>Program part</u>
4.1	Null statement
4.2	Variable declaration
4.3	Assignment statement
4.4	Statement sequence
4.5	WRITE statement
4.6	READ statement

4.1 Null Statement

Although the null statement often only enters programs by accident, it is a legitimate part of CF Pascal. The statement "does nothing," which means that whatever data state exists before its execution is unchanged afterwards. The function with this behavior is the identity *I*. Thus define:

$\boxed{} = I.$

4.2 Variable Declarations

The declarations within a program have a role in the program calculus similar to that of the program header: they modify the data state so that it contains the proper identifier names for the remainder of the program to process. Each VAR declaration has a functional meaning that transforms a data state in which its identifier does not appear, to one in which it does appear. Here we have another choice to make: what value should be associated with such a new-made identifier, and what are the rules for using this value? In some Pascal implementations, a special value that cannot be confused with any other is attached to newly declared identifiers, and this value cannot be referenced without a machine trap, so the variable must be overwritten between declaration and reference. It is more common to attach an arbitrary value to declared identifiers until they are overwritten (often the left-over contents of memory previously used)--such a value *can* be used, with unexpected results. It seems clear that we should adopt the former view: use of a newly declared identifier is a run-time error until it has been given a value.

The effect of a VAR declaration is to extend the domain of the identifier-value map to include a new identifier. To reflect the possibility that any value might be subsequently acquired, we define the meaning of the declaration to be a relation including all values of the appropriate kind. That is,

$$\boxed{\texttt{VAR } V\texttt{: } K} = \{(T, W): W = T \cup \{(V, x): x \text{ is a value of type } K\}\}.$$

This relation in which all possible values are paired with V is awkward to write, so we introduce a shorthand of "?" (not the character ?, as the typography shows) for the multiple values. Then for example the program part

```
VAR Fresh: CHAR
```

transforms execution state

(INPUT*$\underline{A}$BC, OUTPUT*_)

to execution state

(INPUT*$\underline{A}$BC, OUTPUT*_, Fresh*?).

In the box notation:

$$\boxed{\texttt{VAR Fresh: CHAR}}(\texttt{INPUT*}\underline{A}BC, \texttt{OUTPUT*\_})$$
$$= (\texttt{INPUT*}\underline{A}BC, \texttt{OUTPUT*\_}, \texttt{Fresh*?})$$

The declaration of multiple variables requires an obvious extension of this definition.

When a data state is not a function because in it some identifier V is associated with all potential values (in the shorthand, the state contains (V*?)), then the expression V has a meaning that is the undefined function. That is, in this case $\boxed{V}$ is undefined for all states.

4.3 Assignment Statements

The intuitive meaning of the assignment statement as a data-state transformation is that the identifier on the left side ceases to be associated with its old value, and instead becomes associated with a value obtained from the right side. In CF Pascal, assignment statements are of the restricted form:

$V := E$

where V is a variable declared as CHAR and E is either a variable declared as CHAR or a literal character enclosed in single quotation marks, or a nest of SUCC function calls founded on such a variable or literal. Section 3.2 has formally defined the box function for such expressions as a mapping from data states to character values (which may be undefined for some uses of SUCC). The investment in notation now pays off in a concise definition of the meaning of an assignment statement:

$$\boxed{V := E} = \{(T, U)\text{: } U \text{ is the same data state as } T \text{ except that } U(V) = \boxed{E}(T)\}.$$

As usual, the definition includes the implicit case that should $\boxed{E}$ be undefined, then so is the assignment-statement function undefined. The failure in not in the theory's definition of meaning (the box function): the box function *is* defined, to be the everywhere-undefined function.

Here are four examples:

$\boxed{\texttt{V1 := 'C'}}((\texttt{V1}*\mathbb{A}, \texttt{V2}*\mathbb{B})) = (\texttt{V1}*\mathbb{C}, \texttt{V2}*\mathbb{B})$,
$\boxed{\texttt{V2 := V1}}((\texttt{V1}*\mathbb{A}, \texttt{V2}*\mathbb{B}) = (\texttt{V1}*\mathbb{A}, \texttt{V2}*\mathbb{A})$,
$\boxed{\texttt{V1 := SUCC(V1)}}$ is undefined on the state $(\texttt{V1}*\mathbb{Z})$,
$\boxed{\texttt{V2 := V2}}$ is undefined on the state $(\texttt{V2}*?)$.

4.4 Statement Sequences

The fundamental rule of the program calculus is functional composition. To calculate the meaning of a sequence of CF Pascal constructions, first obtain the functional meaning for each one, then compose those functions. In many cases, constructions are separated by a semicolon so we can write:

$$\boxed{S_1;\ S_2}(T) = \boxed{S_2}(\boxed{S_1}(T)),$$

or in the purely functional notation,

$$\boxed{S_1;\ S_2} = \boxed{S_1} \bullet \boxed{S_2},$$

where $\bullet$ indicates composition.

The keywords BEGIN and END are used to group statement lists in Pascal. Within these lists there may be a number of statements (or none). But except for the grouping, there is no meaning attached to the BEGIN-END itself. Its meaning is the meaning of what occurs within it. Thus:

$$\boxed{\texttt{BEGIN } L \texttt{ END}} = \boxed{L}$$

where L is a list of statements.

There is here (and above is less obvious cases) a lack of precision caused by omitting detailed syntactic analysis. Presented with a block of program text surrounded by a box, how is the text to be broken up into units to which the definitions are applied? We have used words like "expression" and "statement" and "declaration" as if they had precise meaning in any such text. And of course, if care were used, they do have precise meaning, given by the derivation tree for the program. In that tree at any level there is precisely one "expression," etc., indicated by the nonterminal of that name in an appropriate CF-Pascal grammar. This use of grammar as a basis for the semantics goes back to ALGOL 60 [1], and may be the most important feature of grammar-based syntax. Here we do not make the correspondence very precise, but in examples we order the compositions as they appear in the derivation tree.

4.5 WRITE-Statements

Whatever a program may do internally to its execution state, the result cannot be observed by a person unless WRITE-statements are included to communicate the internal state to the outside world. In CF Pascal, a WRITE-statement may include only a sequence of expression arguments (of the kind defined in Section 3.2). With this restriction, each argument has a box function already defined. The meaning of the statement can then be given in terms of these components in the natural way: a WRITE-statement appends to the special identifier OUTPUT in the data state the character values of its argument items, in order.

Let a WRITE-statement have arguments E_1, E_2, ... in order. For execution state T, form the values:

$$V_1 = \boxed{E_1}(T),\ V_2 = \boxed{E_2}(T),\ \ldots$$

Then the value attached to OUTPUT in

$$\boxed{\texttt{WRITE(}E_1, E_2, \ldots\texttt{)}}(T)$$

is T(OUTPUT) with V_1 and V_2 and ... appended in order.

Enough of the program calculus has now been presented to handle the very simplest complete programs. For example, if P is:

```
PROGRAM WriteHello(INPUT, OUTPUT);
  VAR
    LetterL: CHAR;
  BEGIN
    LetterL := 'L';
    WRITE('H', 'E', LetterL, LetterL, 'O')
  END.
```

then if the input string is x the program header and VAR declaration establish the state

(INPUT·$\underline{x}$, OUTPUT·_, LetterL·?)

on which the program works as follows:

$$\boxed{\texttt{WRITE('H', 'E', LetterL, LetterL, 'O')}}$$
$$(\boxed{\texttt{LetterL := 'L'}}$$
$$((\texttt{INPUT}\cdot\underline{x}, \texttt{OUTPUT}\cdot\_, \texttt{LetterL}\cdot?)))$$
$$= \boxed{\texttt{WRITE('H', 'E', LetterL, LetterL, 'O')}}$$
$$((\texttt{INPUT}\cdot\underline{x}, \texttt{OUTPUT}\cdot\_, \texttt{LetterL}\cdot\texttt{L}))$$

by the action of the assignment statement and statement composition. The values of the arguments in the WRITE-statement are:

$\boxed{\texttt{'H'}}$ ((INPUT*$\underline{x}$, OUTPUT*_, LetterL*$\mathbb{L}$) = $\mathbb{H}$
$\boxed{\texttt{'E'}}$ ((INPUT*$\underline{x}$, OUTPUT*_, LetterL*$\mathbb{L}$) = $\mathbb{E}$
$\boxed{\texttt{LetterL}}$ ((INPUT*$\underline{x}$, OUTPUT*_, LetterL*$\mathbb{L}$) = $\mathbb{L}$
$\boxed{\texttt{'O'}}$ ((INPUT*$\underline{x}$, OUTPUT*_, LetterL*$\mathbb{L}$) = $\mathbb{O}$

The WRITE-statement thus changes the execution state to:

(INPUT*$\underline{x}$, OUTPUT*$\mathbb{HELLO}$_, LetterL*$\mathbb{L}$)

and the final action of the . following END is to produce the meaning of program *P*:

$\boxed{P}$ (y) = $\mathbb{HELLO}$ for any y.

4.6 READ-Statements

The list of arguments in a CF Pascal READ-statement can consist only of identifiers declared as CHAR variables. READ-statement meaning is easy to give if enough characters are available in the data state (i.e., attached to INPUT as future string). However, should there be too few characters, the READ-statement's meaning function goes undefined. The case of multiple variables in a READ-statement is a straightforward extension of the single-variable case.

Suppose then that a data-state value for INPUT contains at least one character in its future string, say c. In the abbreviated form of a data state, write such a value as

INPUT*$x\ \underline{c}\ y$

where x and y represent the parts (if any) of the value not of interest. (x is the past string; the future string begins with c and ends with y.) Then the meaning of

READ(Cv1)

where the variable is suitably declared CHAR is

$\boxed{\texttt{READ(Cv1)}}$ ((INPUT*$x\ \underline{c}\ y$, ..., Cv1*v, ...))
= (INPUT*$x\ c\ \underline{y}$, ..., Cv1*c, ...)

where the division point between the past and future strings for INPUT now occurs just before y.

As an example, consider the program *P*:

```
PROGRAM Change2(INPUT, OUTPUT);
  VAR
    C2: CHAR;
  BEGIN
    READ(C2);
    WRITE(C2);
    READ(C2);
    WRITE('2')
  END.
```

We work out:

$\boxed{P}$ (ABC)

The program header and the VAR declaration establish the execution state:

$$(\texttt{INPUT}*\underline{\mathbb{A}}\mathbb{BC}, \texttt{OUTPUT}*\_, \texttt{C2}*?).$$

Then the successive statements yield:

$$\begin{aligned}
&\boxed{\texttt{READ(C2); WRITE(C2); READ(C2); WRITE('2')}}\\
&\quad((\texttt{INPUT}*\underline{\mathbb{A}}\mathbb{BC}, \texttt{OUTPUT}*\_, \texttt{C2}*?))\\
&= \boxed{\texttt{WRITE(C2); READ(C2); WRITE('2')}}\\
&\quad((\texttt{INPUT}*\mathbb{A}\underline{\mathbb{B}}\mathbb{C}, \texttt{OUTPUT}*\_, \texttt{C2}*\mathbb{A}))\\
&= \boxed{\texttt{READ(C2); WRITE('2')}}\\
&\quad((\texttt{INPUT}*\mathbb{A}\underline{\mathbb{B}}\mathbb{C}, \texttt{OUTPUT}*\mathbb{A}\_, \texttt{C2}=\mathbb{A}))\\
&= \boxed{\texttt{WRITE('2')}}\,((\texttt{INPUT}*\mathbb{AB}\underline{\mathbb{C}}, \texttt{OUTPUT}*\mathbb{A}\_, \texttt{C2}*\mathbb{B}))\\
&= (\texttt{INPUT}*\mathbb{AB}\underline{\mathbb{C}}, \texttt{OUTPUT}*\mathbb{A2}\_, \texttt{C2}*\mathbb{B}).
\end{aligned}$$

The final . yields:

$$\boxed{P}\,(\mathbb{ABC}) = \mathbb{A2}.$$

Reading past end of file is a runtime error in CF Pascal, so in the definition we must exclude those data states in which there are insufficient characters on the future string attached to INPUT. Define:

$\boxed{\texttt{READ}(C)}$ = {(*T*, *U*): the first character of the future string in *T*(INPUT) is *c*, and *U* is the same as *T* except that (i) *c* is transferred to the end of the past string in *U*(INPUT) and (ii) *U*(*C*) = *c*}.

As usual, note the failure of definition expressed by there being no first character as required, and hence no ordered pair in the defined function with this property.

4.7 File Input-Output

CF Pascal acquires its power from the use of intermediate files which may be written and then read back. For example, sorting can be accomplished by separating the input into multiple streams which can then be merged. For a variable Fv of type TEXT the statements

```
WRITE(Fv, ...)
READ(Fv, ...)
```

act as those defined above, but on the file Fv instead of OUTPUT and INPUT. There is considerable complexity introduced by this extension, because files must be properly initialized with

```
REWRITE(Fv)
```

for (over)writing, and

```
RESET(Fv)
```

for (re)reading. The ability to specify a file name raises the possibility of numerous runtime errors, for example, trying to read a file that is being written, without using RESET. To handle these complications requires adding information to the pair of strings that is a data-state file value, namely a "mode" tag for read/write status. File values are then triples consisting of the past and future strings and the mode tag. The meaning of REWRITE and RESET, and the changes to the meaning of WRITE and READ can then be given in a straightforward way. Since we will not here analyze programs using file input-output, these definitions are omitted.

4.8 Analysis of Linear Programs

The examples of this section have shown that given a linear program and any particular input string, the calculation of the resulting output string (if any) is a straightforward, mechanical process. It is a little more difficult to begin with the program alone, and calculate the function it means. The difficulty is one of notation, since the set-theoretic definition of the meanings of program parts can be difficult to combine concisely. We mention one of several techniques that are useful in practice, the *trace table* [9]. This device can be applied to any series of assignments, and amounts to a symbolic execution of the program. A tabular sequence of equations is created, in which subscripted versions of the variables appear, defined in terms of earlier such variables. The set of equations can be solved for final variable values in terms of the initial ones. For example, the program fragment

```
BEGIN
  V1 := V4;
  V2 := V3;
  V3 := V2;
  V4 := V1
END
```

cannot be understood directly without some effort, but yields easily to the trace-table method:

	V1	V2	V3	V4
`V1 := V4;`	$V1_1=V4_0$	$V2_1=V2_0$	$V3_1=V3_0$	$V4_1=V4_0$
`V2 := V3;`	$V1_2=V1_1$	$V2_2=V3_1$	$V3_2=V3_1$	$V4_2=V4_1$
`V3 := V2;`	$V1_3=V1_2$	$V2_3=V2_2$	$V3_3=V2_2$	$V4_3=V4_2$
`V4 := V1`	$V1_4=V1_3$	$V2_4=V2_3$	$V3_4=V3_3$	$V4_4=V1_3$

These subscripted variables are of the usual mathematical sort, not the program sort; that is, they represent fixed unknowns, and the set of equations can be solved by repeated substitution:

$$V1_4 = V1_3 = V1_2 = V1_1 = V4_0$$
$$V2_4 = V2_3 = V2_2 = V3_1 = V3_0$$
$$V3_4 = V3_3 = V2_2 = V3_1 = V3_0$$
$$V4_4 = V1_3 = V1_2 = V1_1 = V4_0$$

That is, collecting these results:

$$V1_4 = V4_0,\ V2_4 = V3_0,\ V3_4 = V3_0,\ V4_4 = V4_0$$

The analysis has shown that the program above has the same effect as

```
BEGIN
  V1 := V4;
  V2 := V3
END
```

which may be a surprise to the programmer.

5 Correctness

Any program has a purpose, but that purpose may not require results in some exact form. For example, a program may be required to print the members of a set, without their order or the page layout being specified. These variations can be described by providing, for each instance of input data, a set of acceptable instances of output data. The description is a relation consisting of all pairs of acceptable instances of input data and output data. Therefore, a ***program specification*** is defined as any relation whose domain and range are sets of character strings.

Just as a program function may be difficult to describe in a well-known mathematical form, but nevertheless exists for every program; so a program specification may be difficult to set down, but is a mathematical relation nevertheless.

An important special case of a specification occurs when the acceptable pairs of input data and output data form a function; that is, for each instance of input data exactly one instance of output data is acceptable. This special case of a specification relation is therefore a *specification function*.

The specification (relation or function) for a program is a mathematical form of what the program is *supposed* to do, a description of desired results. This form is entirely independent of any program to realize it, and in fact is the starting point for writing a program. It is important to recognize that a specification gives no information about *how* some program might perform to meet it. Since it is simply a collection of input-output pairs, it states *what* is to be done, without a hint of a method for doing it. On the other hand, once a particular program exists, its program function (which is the same kind of mathematical object as a specification function) defines what the program *does* do, without regard for any intentions the programmer may have had. Furthermore, using the program calculus, this meaning can be calculated step-by-step from the program text itself. The program function itself does not express how the program accomplishes what it does, but to calculate the program function requires full details of the program's inner workings. A central question of programming can be simply stated in these terms:

Given a specification and a program, does the program fulfill that specification?

The technical definitions necessary to state this question are already available. Given a program specification relation r and a program P, we say that *P is correct with respect to r* if and only if, for every member x of the domain of r (an instance of input data), P produces some member of the range of r which is paired with x in r. That is, for each input x, P produces result y such that $(x, y) \in r$.

<u>Theorem</u> (Program Correctness)

Program P is correct with respect to specification relation r if and only if:

$$\text{domain}(r \cap \boxed{P}) = \text{domain}(r)$$

<u>Proof</u> The expression $r \cap \boxed{P}$ identifies all acceptable pairs of r computed by P. Therefore domain($r \cap \boxed{P}$) identifies the set of input data for which P produces acceptable output data. Since domain(r) is the set of input data for which r specifies acceptable output data, the condition

$$\text{domain}(r \cap \boxed{P}) = \text{domain}(r)$$

ensures that P produces acceptable output data for every instance of input data defined by r. QED

Note that P may execute successfully for input data not identified by r, but such pairs of $\boxed{P}$ are screened out of $(r \cap \boxed{P})$ by r. Note also that if P produces an unacceptable instance of output data, no member of r with that input data can be in $(r \cap \boxed{P})$, and therefore domain($r \cap \boxed{P}$) cannot coincide with domain(r).

In case the program specification is a function, the condition for program correctness can be simplified as follows:

<u>Corollary</u> (Program Correctness)

Program P is correct with respect to specification function f if and only if

$$f \subseteq \boxed{P}.$$

<u>Proof</u> The expression $f \cap \boxed{P}$ identifies all acceptable pairs of f computed by P, which must be f, itself. That is, P

is correct with respect to f if and only if

$$f \cap \boxed{P} = f,$$

and thus if and only if $f \subseteq \boxed{P}$. QED

6 Conditional Statements

Much of the power (and complication) in programs comes from their conditional statements, which provide the means of making decisions based not only on program input, but on intermediate values internal to the program. However, the meaning of a single conditional in isolation is easy to define.

6.1 Meaning of Conditionals

Let

```
S = IF B
    THEN
     T
    ELSE
     E
```

where B is a Boolean expression and T, E are statements. Then

$$\boxed{S}(U) = \begin{cases} \boxed{T}(U) \text{ if } \boxed{B}(U) \\ \boxed{E}(U) \text{ if } \sim\boxed{B}(U) \\ \text{otherwise } \boxed{S} \text{ is not defined at } U. \end{cases}$$

For example if

```
S1 = IF V1 < V2
       THEN
         V1 := V2
       ELSE
         V2 := V1
```

Then

$$\boxed{S_1}((\text{V1}*\text{A}, \text{V2}*\text{B})) = (\text{V1}*\text{B}, \text{V2}*\text{B})$$

because

$$\boxed{B}((\text{V1}*\text{A}, \text{V2}*\text{B})) = \underline{\text{true}},$$
$$\boxed{T}((\text{V1}*\text{A}, \text{V2}*\text{B})) = (\text{V1}*\text{B}, \text{V2}*\text{B}),$$

and the value of $\boxed{S_1}$ for this data state is given by the value of $\boxed{T}$.

In order to give a definition by cases without explicitly naming the data state, it is necessary to select one of two sets of ordered pairs according to the Boolean expression. The following standard trick accomplishes this:

$$\boxed{S} = \{(U, \boxed{T}(U)) : \boxed{B}(U)\}$$
$$\cup \{(U, \boxed{E}(U)) : \sim \boxed{B}(U)\}.$$

The first set contains all state pairs in which the condition holds, and the second set those pairs in which it does not hold. It is important to note the way that failure of definition can occur here. There is no "evaluation" of these sets in any order. They simply contain or fail to contain certain pairs. For example, should $\boxed{E}$ fail to be defined for some U, that U will not occur in the second set, independent of $\boxed{B}$. Such a failure has no influence whatsoever on pairs in the first set. However, failure of $\boxed{B}$ is another matter. If this function fails to be defined for some U, neither condition holds, and U is not paired with anything in either set; that is, $\boxed{S}$ is undefined for such a U.

The IF-statement

```
IF B
 THEN
  T
```

could be given a similar definition, but it can also be agreed to mean the same as:

```
IF B
 THEN
  T
 ELSE
   {null statement}
```

so that its definition can be derived as follows:

$$\boxed{\text{IF } B \text{ THEN } T}(U) = \begin{cases} \boxed{T}(U) \text{ if } \boxed{B}(U) \\ U \text{ if } \sim \boxed{B}(U) \\ \text{otherwise undefined} \end{cases}$$

or, in the functional notation:

$$\boxed{\text{IF } B \text{ THEN } T} =$$
$$\{(U, V) : \boxed{B}(U) \text{ and } V = \boxed{T}(U)\}$$
$$\cup \{(U, U) : \sim \boxed{B}(U)\}.$$

In the second case, the identity function is applied to the state, since that is the meaning of the null statement.

6.2 Analysis of Conditional Statements

Composing the functions that result from IF-statements is no more difficult in principle than composing imperative-statement functions, but the notational complication is even more severe. The trace table can be extended to help in practical cases. A conditional trace table has an additional column of conditions expressed in terms of the values in the table, namely the conditions required for the assignments in the table to take place. For example, the program part S:

```
BEGIN
  IF V1 < V2
    THEN
      V1 := V2
    ELSE
      V2 := V3;
  V1 := V3;
  IF V2 < V3
    THEN
      V2 := V3
    ELSE
      V3 := V1
END
```

will execute in one of four sequences, depending on which of the THEN or ELSE statements of the two IF-statements are selected. Identify these sequences by noting T or E for the THEN or ELSE alternatives in order, so *S*(T,T) means the two THEN parts are taken, while *S*(E,T) means that the first ELSE is taken, then the second THEN. This gives a conditional trace table such as:

S(T,T)	Condition	V1	V2
`IF V1 < V2`	$V1_0 < V2_0$		
`THEN V1 := V2`		$V1_1 = V2_0$	
`V1 := V3`	true	$V1_2 = V3_1$	
`IF V2 < V3`	$V2_2 < V3_2$		
`THEN V2 := V3`			$V2_3 = V3_2$

The subscripts of a condition refer to the values of the previous line, since the condition is evaluated at the beginning of the statement. An ordinary assignment has the condition true. For this particular sequence to be executed, every condition must hold, so the condition for the sequence is the conjunction of the conditions in the table. Eliminating subscripts, this condition is:

$$(V1_0 < V2_0) \text{ and } \underline{\text{true}} \text{ and } (V2_0 < V3_0)$$

(since $V2_2 = V2_0$ and $V3_2 = V3_0$ from the trace table) and the equations that result are:

$$V1_3 = V3_0,\ V2_3 = V3_0,\ V3_3 = V3_0.$$

Thus the meaning in the case *S*(T,T) is the set

$$\{(U,\ \boxed{\texttt{V1 := V3; V2 := V3}}\,(U)):\ \boxed{\texttt{V1 < V2}}\,(U) \text{ and } \boxed{\texttt{V2 < V3}}\,(U)\}.$$

Similarly, we find

S(T,E)	Condition	V1	V3
`IF V1 < V2`	$V1_0 < V2_0$		
`THEN V1 := V2`		$V1_1 = V2_0$	
`V1 := V3`		$V1_2 = V3_1$	
`IF V2 < V3`	$V2_2 \geq V3_2$		
`ELSE V3 := V1`			$V3_3 = V1_2$

from which we deduce the condition:

$$(V1_0 < V2_0) \text{ and } (V2_0 \geq V3_0)$$

and the equations:

$V1_3 = V3_0, V3_3 = V3_0$

so that the meaning in this case is:

$$\{(U, \boxed{\texttt{V1 := V3}}(U)): \\ \boxed{\texttt{V1 < V2}}(U) \text{ and } \boxed{\texttt{V2 >= V3}}(U)\}.$$

The case S(E,T) cannot occur (the conditional trace table leads to an empty meaning set); a similar analysis for case S(E,E) yields a third set, and their union is the meaning of S:

$$\begin{aligned}\boxed{S} = &\{(U, \boxed{\texttt{V1 := V3; V2 := V3}}(U)): \\ &\quad \boxed{\texttt{V1 < V2}}(U) \text{ and } \boxed{\texttt{V2 < V3}}(U)\} \\ \cup &\{(U, \boxed{\texttt{V1 := V3}}(U)): \\ &\quad \boxed{\texttt{V1 < V2}}(U) \text{ and } \boxed{\texttt{V2 >= V3}}(U)\} \\ \cup &\{(U, \boxed{\texttt{V1 := V3; V2 := V3}}(U)): \\ &\quad \boxed{\texttt{V1 >= V2}}(U)\}.\end{aligned}$$

7 Iteration Statements

With iteration-free programs, we have seen how to derive program meaning as a composition of the meaning of program parts, in which the number of parts is determined by the static program text. However, with iteration, program parts can be executed repeatedly. If the number of iterations were fixed, the part functions would be fixed compositions of simpler part functions. But the great power of iteration statements arises from a variable number of iterations, so we need not be surprised that the difficulty of dealing with iteration statements increases accordingly.

Although in a long iteration-free program there could be a great deal of notational difficulty in calculating the program's meaning function, there is no difficulty in principle: each statement has its functional meaning, and the meaning of the whole is simply their composition. There is no mechanical way to deal similarly with iteration statements, but this section presents a definition of meaning, and techniques for proving that a loop has a certain meaning, which must be (nonmechanically) guessed or supplied independent of the program text.

7.1 Meaning of Iteration Statements

The power of iteration exacts a high price when it comes to calculating the meaning of WHILE-statements. The pattern of our definitions has been to give the function of each statement-type in terms of its parts. The parts of a WHILE-statement are evidently the condition (determining if the iteration should continue) and the loop body (what to do if it should continue). The difficulty is that the action of the loop body is repeated in forming the meaning of the loop as a whole, and this repetition occurs a number of times that is not explicit in the text. If the number of times k the iteration occurs were known, there would be a way out of the difficulty, for then the function of the entire WHILE-statement would be a composition of its body's function exactly k times. For example, if it were magically given that

WHILE *B* DO *D*

"went around" exactly twice, then we could consider it equivalent to

BEGIN *D*; *D* END

and the result would be

$\boxed{\text{WHILE } B \text{ DO } D} = \boxed{D;\ D}$.

There is yet a further complication in defining the semantics of the WHILE-statement. It may happen that the loop never terminates. In that case the "number of iterations" makes no sense, and the function of the loop is undefined for the state that caused the unlimited repetition. The number of iterations to completion is in all cases the key to the WHILE-statement. If this number is *k*, then the function of the loop is *k*-fold composition of the function for the loop body; if the iteration continues without end, the function is undefined.

There is a direct way to capture the function of a loop, by asserting that the state resulting from the loop's execution is the result of the loop body executing *k* times, for some unspecified *k*. To be consistent with the intuitive meaning of the loop, this *k* (if it exists) has the property that the loop condition is true before the 1st, 2nd, ... *k*th iteration, then is false. That is, after *k* iterations the condition fails for the first time. The WHILE-statement meaning therefore consists of exactly those ordered pairs for which there is the appropriate *k*, with the output state being a *k*-fold action of the loop body on the input state.

The definition of meaning along these lines will now be given. Let WHILE-statement *W* be

```
WHILE B DO
  D
```

where *B* is a Boolean expression and *D* is a statement. Then define

$$
\begin{aligned}
\boxed{W} = \{(T, U):\ & \exists k \text{ integer}, k \geq 0 \\
& \forall i \text{ integer}, 0 \leq i < k \\
& \quad (\boxed{B}(\boxed{D}^{i}(T)) \\
& \text{and} \\
& \quad \sim \boxed{B}(\boxed{D}^{k}(T)) \\
& \text{and} \\
& \quad \boxed{D}^{k}(T) = U)\}.
\end{aligned}
$$

For example, consider the loop *U*:

```
WHILE V1 > '1' DO
  IF V1 = '8' THEN
    V1 := '0'
  ELSE
    IF V1 = '9' THEN
      V1 := '1'
    ELSE
      V1 := SUCC(SUCC(V1))
```

Suppose that the value attached to V1 in the input state for *U* is *x*. How many times *k* will the loop be repeated? If $x \leq 1$, then $k = 0$. If *x* is a digit, then $k = (10\text{-}x)/2$ rounded up to the nearest integer. (For example, if $x = 9$, then $(10\text{-}9)/2 = 1/2$ or rounded up, $k = 1$; if $x = 3$, then $(10\text{-}3)/2 = 7/2$, or $k = 4$, etc. It is clear, however, that V1 will end up 1 or 0, without the necessity of determining *k* exactly.) If $x > 9$, after a number of iterations roughly half the distance from *x* to the end of the character-order subsequence, the meaning of the SUCC expression will be the undefined function. This in turn causes the assignment statement to mean the undefined function, and this means that in the definition of the WHILE-statement there is no *k* for the case $x > 9$. Thus the meaning of the loop *U* is:

$$
\begin{aligned}
\boxed{U} = \{(T, T)\colon\ & T(\mathtt{V1}) \le 1\} \\
\cup\ \{(T, A)\colon\ & 1 < T(\mathtt{V1}) \le 9 \text{ and} \\
& A \text{ is the same as } T \text{ except} \\
& \text{that } A(\mathtt{V1}) \text{ is } 1 \text{ or } 0 \text{ as} \\
& T(\mathtt{V1}) \text{ is an odd or even digit}\}.
\end{aligned}
$$

7.2 Analysis of Iteration Statements

When the number of iterations cannot be easily determined, the loop controlled by the WHILE-statement cannot be easily unwound as its loop body acting over and over. But it can always be unwound into the first time the body acts, and then the rest of the times, if that first execution is guarded by a test to cover the case that the loop doesn't execute at all. That is,

```
WHILE B DO D
```

is equivalent to

```
BEGIN  IF B THEN D;  WHILE B DO D END.
```

The equivalence can be seen by examining cases. First, suppose the original loop body *D* is never executed (because condition *B* fails immediately). Then in the expanded version the IF condition *similarly fails*, so the broken-out body is not executed, then in the repetition of the loop itself the condition B fails again, so the body is again not done. Thus the two programs agree in this case. Second, suppose the original loop in fact executed its body exactly once. Then the condition initially succeeds, but something in the body causes it to fail when tried a second time. In the expanded version exactly the same behavior is observed, with the IF condition succeeding, the broken-out body executing and the repeated WHILE condition then failing. The remaining cases in which the original loop executed more than once are similar; the broken-out body takes the first execution, and the repeated loop picks up the remainder.

The discussion in the preceding paragraph can be mirrored exactly in the formal meanings we have defined, where the functional meaning for each of the code fragments can be determined, and "equivalence" means that the functions are the same. The analysis by cases becomes a formal induction on the number of iterations required for the loop to terminate.

It may seem surprising that this simple device can help with the analysis of loops, but it does. The reason is that the repeated loop is exactly the same loop as the original, and this allows us to write an equation in which the loop function occurs twice. Equating the meaning functions for the two loops:

$$
\begin{aligned}
&\boxed{\mathtt{WHILE}\ B\ \mathtt{DO}\ D} \\
&\quad = \boxed{\mathtt{BEGIN}\ \ \mathtt{IF}\ B\ \mathtt{THEN}\ D;\ \ \mathtt{WHILE}\ B\ \mathtt{DO}\ D\ \mathtt{END}}\ .
\end{aligned}
$$

In the compound statement of the second line the first part is a conditional, and the meaning of that can be worked out separately:

$$
\begin{aligned}
&\boxed{\mathtt{WHILE}\ B\ \mathtt{DO}\ D} \\
&\quad = \boxed{\mathtt{IF}\ B\ \mathtt{THEN}\ D} \bullet \boxed{\mathtt{WHILE}\ B\ \mathtt{DO}\ D}\ .
\end{aligned}
$$

The function of the loop has explicitly reentered the equation. It will be clearer if we name this function so it can be easily recognized. Let

$$f = \boxed{\mathtt{WHILE}\ B\ \mathtt{DO}\ D}$$

Then the equation above is

$f = \boxed{\texttt{IF } B \texttt{ THEN } D} \circ f,$

or

$f(T) = f(\boxed{\texttt{IF } B \texttt{ THEN } D}(T)).$

This recurrence equation in function f is useful because very often we can guess (or believe a comment to discover) what a program is supposed to do. Then to prove that it does indeed do so, the guessed function can be substituted into the equation for f. Care is required in this operation, however. The recurrence equation is one that the function f for the loop must satisfy; but it does not follow that any function satisfying the equation is in fact the function of the loop. A simple example will show the pitfall. Consider a loop that never terminates:

```
WHILE 'A' = 'A' DO {nothing}.
```

It is clear that this WHILE-statement has a function that is empty--it contains no ordered pairs because the loop does not terminate on any input. But the recurrence reduces to

$f = f$

for this loop, since

$\boxed{\texttt{IF 'A' = 'A' THEN \{nothing\}}}$

is the identity function. *Any* function f satisfies $f = f$, yet all save the empty function are wrong for the loop.

The remedy for this problem is to add conditions which rule out such extraneous solutions to the recurrence equation. One obvious condition, in view of the pitfall above, is to ensure that the function is not defined when the loop fails to terminate. That is, for f to be the loop function requires that

$\text{domain}(f) \subseteq \text{domain}(\boxed{\texttt{WHILE } B \texttt{ DO } D}).$

Another pitfall is shown by the loop:

```
WHILE 'A' <> 'A' DO {nothing}.
```

In this case the WHILE-statement has a function that is the identity function; it acts as a null statement. But, again, since

$\boxed{\texttt{IF 'A' <> 'A' THEN \{nothing\}}}$

is the identity function, the recurrence equation reduces to

$f = f$

and any function satisfies the equation. All but the identity function are wrong for the loop, however. Many functions satisfy the domain constraint developed above as well; but, all except the identity function violate another easily checked condition when B does not hold:

$f(T) = T$ whenever $\sim \boxed{B}(T).$

Happily, just these two additional conditions are sufficient to ensure that an f satisfying the recurrence equation is the function of the WHILE-statement.

<u>Theorem</u> (WHILE-statement Verification)

Let

$W = \text{WHILE } B \text{ DO } D.$

Then

$f = \boxed{W}$

if and only if:

1. $\text{domain}(f) \subseteq \text{domain}(\boxed{W})$
2. $f(T) = T$ whenever $\sim \boxed{B}(T)$
3. $f(T) = f(\boxed{\text{IF } B \text{ THEN } D}(T))$.

Proof. First, suppose $f = \boxed{W}$. Then conditions 1.-3. must be established.

1. $\text{domain}(f) \subseteq \text{domain}(\boxed{W})$ because f and $\boxed{W}$ are the same function.

2. Suppose $\sim \boxed{B}(T)$ for some T. Then $\boxed{W}(T) = T$ by definition, and hence since $f = \boxed{W}$, we have $f(T) = T$ as required.

3. It has been argued above that

$$\boxed{\text{WHILE } B \text{ DO } D}$$
$$= \boxed{\text{IF } B \text{ THEN } D\text{; WHILE } B \text{ DO } D}$$

and by definition of composition and W this is

$$\boxed{W}(T) = \boxed{W}(\boxed{\text{IF } B \text{ THEN } D}(T)),$$

so the same equation follows for f, since $f = \boxed{W}$.

Conversely, suppose

1. $\text{domain}(f) \subseteq \text{domain}(\boxed{W})$
2. $f(T) = T$ whenever $\sim \boxed{B}(T)$
3. $f(T) = f(\boxed{\text{IF } B \text{ THEN } D}(T))$.

Then, we will show that

$$f = \boxed{W}.$$

Let T be any member of domain(f). Then, by 1., T is a member of $\text{domain}(\boxed{W})$. That is, $\boxed{\text{WHILE } B \text{ DO } D}$ is defined for input state T. Therefore, by definition there exists a k (depending on T) such that

$\boxed{B}(\boxed{D}^k(T))$ is false,

but for each $0 \leq i \leq k\text{-}1$,

$\boxed{B}(\boxed{D}^i(T))$ is true.

Then in hypothesis 3, substitute k-1 times for f:

$$\begin{aligned} f(T) &= f(\boxed{\text{IF } B \text{ THEN } D}(T)) \\ &= f(\boxed{\text{IF } B \text{ THEN } D}(\boxed{\text{IF } B \text{ THEN } D}(T))). \\ &= \ldots \\ &= f(\boxed{\text{IF } B \text{ THEN } D}^k(T)) \end{aligned}$$

using the associativity of composition. From the definition of the conditional, this is

$f(T) = f(\boxed{D}^k(T))$

since each of the evaluations is at a state where $\boxed{B}$ is <u>true</u>. On the right side of this equation, the state is one in which $\boxed{B}$ is <u>false</u>, so by hypothesis 2, f does not alter this state. Thus

$f(T) = \boxed{D}^k(T)$,

and by definition of the loop terminating after k iterations,

$\boxed{W} = \boxed{D}^k$,

so $f = \boxed{W}$. QED

For example, consider the WHILE-statement U of the last section:

```
WHILE V1 > '1' DO
   IF V1 = '8' THEN
     V1 := '0'
   ELSE
     IF V1 = '9' THEN
       V1 := '1'
     ELSE
       V1 := SUCC(SUCC(V1))
```

whose function was claimed to be:

$f = \{(T, T): T(\text{V1}) \leq 1\}$
$\quad \cup \{(T, A): 1 < T(\text{V1}) \leq 9$ and
$\qquad A$ is the same as T except
$\qquad$ that $A(\text{V1})$ is 1 or 0 as
$\qquad T(\text{V1})$ is an odd or even digit$\}$.

To prove this using the WHILE-statement verification theorem we must show:

1. domain$(f) \subseteq$ domain$(\boxed{U})$
2. $f(T) = T$ whenever $\sim \boxed{\text{V1 > '1'}}(T)$
3. $f(T) = f(\boxed{\text{IF V1 > '1' THEN } D}(T))$,

where D is:

```
IF V1 = '8' THEN
  V1 := '0'
ELSE
  IF V1 = '8' THEN
    V1 := '1'
  ELSE
    V1 := SUCC(SUCC(V1))
```

1. The domain of f is evidently $\{c: c \leq 9\}$. The program evidently terminates on V1-values of 8 and 9, and for values of V1 less than 1. On the remaining digits the program advances along the sequence toward 9 or 8 by two steps, and so must halt.

2. Immediate from the definition of f.

3. Consider two cases for the data state T:

If $T(\text{V1}) \leq 1$, then the box function of the conditional is the identity, and 3. holds.

If $T(\text{V1}) >$ [illegible], then the conditional box function is just $\boxed{D}$, so we require $f(T) = f(\boxed{D}(T))$, and this is evidently so by an analysis of the cases [illegible], [illegible], and [illegible], since the double SUCC preserves even- and oddness.

This calculation of the meaning of a simple WHILE-statement illustrates a difference in the program calculus from the calculations of Sections 4 and 6. For iteration-free code, the meaning of programs can be mechanically calculated, using the rules given in those sections. For WHILE-statements things are not so nice: it is necessary to be given or to *guess* the meaning, then verify that the guess is correct. The situation is appropriate to the power of iteration. In practice, finding a trial function on which to use the WHILE-statement verification theorem is not a problem. A helpful comment often supplies one, or intuitive understanding of the code can be used to work one out. Once a guess is in hand, the program calculus establishes that it is or is not correct, with rigor equal to that used to derive the functions of the simpler constructions.

8 Procedures

The meaning of CF Pascal procedure-call statements should be easy to define, since a procedure body consists of statements whose meaning has already been given. (If the body contains procedure calls, the definition should close at this point.) Three ideas complicate the picture:

(1) Procedures have local variables, whose names may conflict with other variables in the program. They may make use of global variables from an environment different than the one existing at the point of call. In technical terms, the data state for a procedure's body may be quite different from the data state existing before and after its call.

(2) Procedures have *parameters* (called by strict reference in CF Pascal), which behave partly as local variables subject to the difficulties of (1) above, and partly as links into the calling environment. In the latter role the problem of aliasing must be handled: apparently distinct parameter variables may be a single called variable.

(3) Procedures may be called recursively, introducing repeated instances of problems of (1) and (2) above, and the further difficulty that the meaning of a call may be defined in terms of another call on the same procedure.

These complications can be handled by small changes in the data-state notation, and by a device based on the ALGOL 60 "copy rule" [1].

In outline, the meaning of a procedure-call statement is the meaning of the procedure declaration. The procedure header plays a role similar to that played by the program header: it transforms the data state from the one at call to the one needed for the body, and after the body's meaning has been obtained, the calling data state is restored. Some adjustments are necessary to handle variable conflicts between the calling and called data state. But except for these technical details, the bulk of procedure-call meaning is simply the meaning of the statements in the declaration, most of which are those defined in Sections 4, 6, and 7. When a procedure call occurs within a procedure body there is no difficulty--the definition is simply applied again and eventually a lowest level is reached in which there are no more calls--unless the call is recursive.

8.1 Procedure Statement Meaning

The meaning of a procedure call in a data state T (the calling state) is the meaning of the procedure's declaration in that state. Within a procedure declaration the only syntax that has no defined meaning (once procedure statements have been defined) is the procedure header. We let the meaning of the header be a mapping that properly alters the data state to account for the procedure's parameters. Imagine that a variable A is passed as actual parameter for a formal parameter X. Then we wish to augment the calling data state by attaching X to whatever value A is attached to there, and maintain this identification of X and A so long as the called procedure is active. A good notation pairs these two identifiers with the value. For example, if the procedure declaration were

```
PROCEDURE Pro(VAR P1: CHAR)
   . . .
```

and were called

```
Pro(Val)
```

in the calling data state

(..., Val*ξ, ...),

then the modified state would be

(..., (Val, P1)*ξ, ...).

Formally, this change is difficult to make. The data state is no longer a mapping from identifiers to values. If its domain is instead taken to be *sets* of identifiers, and *V* occurs in one such set in state *T*, then the definitions must be adjusted so that *T*(*V*) is the value attached to that set. Similarly, in the meaning of assignment and READ-statements, values must be attached to the sets in which the identifier acquiring a value occurs. Without making the formal changes to data states, we will use the notation in which sets of identifiers are associated with a value, and suppose that the box function of an identifier results in the value attached to the proper set.

For simplicity in defining the meaning of a procedure header, consider a single parameter. If the declaration is

```
PROCEDURE Rou(V);
  B
```

where *B* is the complete text of the definition, then the meaning of a procedure statement

```
Rou(A)
```

is defined to be:

$$
\begin{aligned}
&\boxed{\texttt{Rou}(A)}\\
&= \boxed{\texttt{PROCEDURE Rou}(V)\texttt{; } B}\\
&= \boxed{\texttt{PROCEDURE Rou}(V)} \circ \boxed{B}\\
&\qquad \circ \boxed{\texttt{PROCEDURE Rou}(V)}^{-1}.
\end{aligned}
$$

The header meaning

$$\boxed{\texttt{PROCEDURE Rou}(V)}\,(T)$$

is a state *U* the same as *T* except that *V* is paired with *A* in its identifiers, both taking the value $\boxed{A}\,(T)$ in *U*. The inverse of this mapping undoes this data-state transformation: the value that was attached to the set containing *V* and *A* is restored to *A* alone. Thus, the meaning of a procedure call is to alter the calling data state to include the parameters, carry out the meaning of the procedure's statements, then restore the calling identifiers, some of whose values may have changed.

8.2 Identifier Conflicts and Local Variables

In pathological cases, the parameter identifiers added to the calling state by the procedure header may conflict with identifiers already in that state. Should this happen, it is the former that should give way to establish the meaning of CF Pascal. Similarly, within the body of a procedure there are local VAR declarations, and according to the semantics given in Section 4.2 these modify the state. Should there be a conflict in identifiers, these local variables must give way as well. A mechanism for resolving identifier conflicts was invented for just this purpose in ALGOL 60, as a part of the "copy-rule" definition of procedure meaning, and we adopt it.

Whenever a state is altered in defining the meaning of a procedure, each identifier to be added is checked against those already present in the state. Should there be a conflict, the additional identifier is systematically replaced. New identifiers are created by appending i to the original one until a nonconflicting name is created. This new identifier is then substituted throughout the text for the original, before the state is changed. Thus the new, unique identifier enters the state, and the program text whose meaning is being defined contains that new identifier whenever it should to preserve the original intent.

A notation for the systematic substitutions required to avoid identifier conflicts is more trouble than it is worth. We will assume the necessary changes have been made, and make them in examples. Where possible the original choice of identifiers will be made to avoid conflict. (But in recursive procedure calls this may be impossible; see Section 8.3.)

The meaning for VAR declarations given in Section 4.2 adjusts the data state to add the newly declared identifier. In the VAR declaration of a PROGRAM there is no need to later remove this identifier, because it ceases to have meaning only when the program is complete, and there the terminating period extracts only the file strings from the data state. However, the VAR declarations within procedures are different. Their identifiers must exist only as a part of the meaning of a call, must not persist following the call. The relation given as meaning for a VAR declaration has an inverse with just the properties needed: it maps a state containing the new VAR and its value (if any) back to one in which the variable does not appear. Thus for a declaration followed by a block:

```
D = VAR V
    BEGIN
       ...
     END
```

the meaning is

$$\boxed{D} = \boxed{\text{VAR } V} \bullet \boxed{\text{BEGIN} \ldots \text{END}} \bullet \boxed{\text{VAR } V}^{-1}$$

As an example of a procedure call, consider the program:

```
PROGRAM Pr2(INPUT, OUTPUT);
  VAR
    Next: CHAR;
  PROCEDURE CollectV(Flag: CHAR);
    VAR
      Next: CHAR;
    BEGIN
      Next := 'B';
      Flag := 'T'
    END;
  BEGIN  {Pr2}
    Next := 'A';
    CollectV(Next)
  END.
```

When the procedure call that ends Pr2 occurs, the data state (for input string x) is:

(INPUT•x, OUTPUT•_, Next•A).

Since this example is not concerned with INPUT and OUTPUT, in the sequel they will be omitted from the state to simplify the notation.

The call on CollectV has the meaning:

$$
\begin{aligned}
&\boxed{\texttt{CollectV(Next)}}\,((\texttt{Next}*\mathbb{A}))\\
&= \boxed{\texttt{PROCEDURE CollectV(Flag: CHAR); VAR ... END}}\,((\texttt{Next}*\mathbb{A}))\\
&= \boxed{\texttt{PROCEDURE CollectV(Flag: CHAR)}}^{-1}\\
&\qquad(\boxed{\texttt{VAR Next: CHAR; BEGIN ... END}}\,(((\texttt{Next, Flag})*\mathbb{A})))\\
&= \boxed{\texttt{PROCEDURE CollectV(Flag: CHAR)}}^{-1}(\boxed{\texttt{VAR Nexti: CHAR}}^{-1}\\
&\qquad(\boxed{\texttt{BEGIN Nexti := ... END}}\\
&\qquad\qquad(((\texttt{Next, Flag})*\mathbb{A}, \texttt{Nexti}*?)))).
\end{aligned}
$$

Working out the meaning of the body itself is straightforward:

$$
\begin{aligned}
&\boxed{\texttt{BEGIN Nexti := ... END}}\\
&\qquad\qquad(((\texttt{Next, Flag})*\mathbb{A}, \texttt{Nexti}*?))\\
&= ((\texttt{Next, Flag})*\mathbb{T}, \texttt{Nexti}*\mathbb{B}),
\end{aligned}
$$

so the result is

$$
\begin{aligned}
&\boxed{\texttt{PROCEDURE CollectV(Flag: CHAR)}}^{-1}(\boxed{\texttt{VAR Nexti: CHAR}}^{-1}\\
&\qquad(((\texttt{Next, Flag})*\mathbb{T}, \texttt{Nexti}*\mathbb{B})))\\
&= (\texttt{Next}*\mathbb{T}).
\end{aligned}
$$

Here is an example program *P* on which many students of Pascal (and many early compiler-writers as well) have foundered:

```
PROGRAM Confused(INPUT, OUTPUT);
  VAR
    Which: CHAR;
  PROCEDURE Zap;
    BEGIN
      Which := 'Z'  {Zap it, but which?}
    END;
  PROCEDURE Inside;
    VAR
      Which: CHAR;
    BEGIN
      Which := 'B';
      Zap;
      WRITE(Which)
    END;
  BEGIN {Confused}
    Which := 'A';
    Inside;
    WRITE(Which)
  END.
```

This program "Zap"s one of its Which VARs, but which one? Is the meaning

$\boxed{P}\,(\iota) = \mathrm{ZA}$

or

$\boxed{P}\,(\iota) = \mathrm{BZ}$?

(The answer can be guessed by imagining a change of VAR names. Suppose the Which within PROCEDURE Zap were some different name. Then Zap would not have correct syntax: this different name would not be declared, and VAR names within Inside would be of no help. This goes along with the meaning being $\boxed{P}$ (*t*) = 𝔹ℤ.)

Using the definition at the point of the call on Inside:

$\boxed{\texttt{Inside}}$ ((OUTPUT*_, Which*𝔸))
= $\boxed{\texttt{PROCEDURE Inside}}^{-1}$($\boxed{\texttt{VAR Whichi: CHAR; ... END}}$
  ($\boxed{\texttt{PROCEDURE Inside}}$ ((OUTPUT*_, Which*𝔸)))).

Both $\boxed{\texttt{PROCEDURE Inside}}$ and its inverse are the identity function, since there are no parameters, so we have:

= $\boxed{\texttt{VAR Whichi: CHAR}}^{-1}$($\boxed{\texttt{BEGIN Whichi := 'B'; Zap; ... END}}$
  ((OUTPUT*_, Which*𝔸, Whichi*?))))
= ...
= $\boxed{\texttt{VAR Whichi: CHAR}}^{-1}$($\boxed{\texttt{Zap; WRITE(Whichi) END}}$
  ((OUTPUT*_, Which*𝔸, Whichi*𝔹)))).

Because Zap has neither parameters nor VARs, the data state remains the same when its body meaning is applied:

= $\boxed{\texttt{VAR Whichi: CHAR}}^{-1}$
  ($\boxed{\texttt{WRITE(Whichi)}}$ ($\boxed{\texttt{BEGIN Which := 'Z' END}}$
    ((OUTPUT*_, Which*𝔸, Whichi*𝔹))))
= ...
= $\boxed{\texttt{VAR Whichi: CHAR}}^{-1}$
    ((OUTPUT*𝔹, Which*ℤ, Whichi*𝔹))
= (OUTPUT*𝔹_, Which*ℤ).

From here it is easy to calculate that $\boxed{P}$ (*t*) = 𝔹ℤ.

8.3 Recursion

When a procedure call occurs within a procedure there are just two possibilities: either this call and others that it may lead to eventually come down to a procedure body in which no calls occur; or, one of the procedures in the potential chain of calls has occurred previously in the chain. These alternatives are a consequence of the finite nature of programs: unless one of the procedures is recalled, the list of all possibilities must soon be exhausted. When a procedure ends up being reinvoked, it is said to be called recursively.

The consequence of blindly applying the definitions given earlier in this section to a nonrecursive call sequence is happy: a definition of the meaning of the whole program results without difficulty. Similar application to a recursive sequence is less satisfactory: it results in a recurrence relation in which the function computed by the recursive procedure is defined in terms of itself. As a simple example, consider the program:

```
PROGRAM TestOdd(INPUT, OUTPUT);
  {Reads one character from INPUT and returns Y(es) or N(o)
   depending on whether or not it is odd.  The action is not
   defined if INPUT is empty or the first character is not
   a digit.}
  VAR
    One: CHAR;

  PROCEDURE Odd(Result, Val: CHAR);
    VAR
      NextVal: CHAR;
    BEGIN
      IF Val = '9' THEN
        Result  := 'Y':
      ELSE
        IF Val = '8' THEN
          Result := 'N'
        ELSE
          BEGIN
            NextVal := SUCC(SUCC(Val));
            Odd(Result, NextVal)
          END
    END {Odd}

  BEGIN
    READ(One);
    Odd(One, One);
    WRITE(One)
  END.
```

Applying the rules presented above to the call

```
Odd(One, One)
```

near the end gives

$\boxed{\texttt{Odd(One, One)}}$ (T)

T except that One has value Y if *T*(One) = 9
= *T* except that One has value N if *T*(One) = 8
$\boxed{\texttt{PROCEDURE Odd(...)}}^{-1}$($\boxed{\texttt{Odd(Result, NextVal)}}$ (U)) otherwise,

where *U* is a state in which Result and Val are both grouped with One from *T*, and with NextVal added, having the value two characters in sequence beyond the value attached to One in *T*.

This form is reminiscent of the situation that arose in defining the meaning of a WHILE-statement: the meaning is given in terms of itself (using a modified state), but with additional cases that are not recursively defined, "protecting" the recursive case. For the WHILE-statement the protection came from the loop test, and the form of the recurrence was simply related to the loop body; here these elements depend entirely on the form of the procedure body. That is, the case of recursive procedure calls is much less standardized than the case of WHILE-statements.

Given any particular data vector *T*, the definition for the meaning of a recursive call can be "unwound" until the recursion terminates, if that happens. And if it does not terminate, beginning the process of unwinding may expose that fact. For example,

$\boxed{\texttt{Odd(One, One)}}$ (One•6) = (One•N)

comes from one more expansion than shown above. However, solving the recurrence equation in the general case is more difficult than it was for WHILE-statements.

Imagine substituting a guess for the meaning of Odd into the recurrence equation. The parameters play special roles, literally that of parameters, in that the data-state transformation is different when they assume different values. If we believe the comments on the program, the meaning of Odd for parameters (identifiers) x and y is the function f:

f = {(T, U): U = T, except that
        $U(x)$ =   if $T(y)$ is an odd digit, and
        $U(x)$ = [illegible] if $T(y)$ is an even digit}.

(The implication is that if $T(y)$ is not a digit at all, there is no second element of the pair in this set: that is, f is undefined for such T.) Substituting f in the recurrence equation, the left side is:

    T except that One has value   (or [illegible])
        if T(One) is odd (or even).

On the right, the first two cases are in agreement with this data state. For the third case, a digit value must be prior to   in sequence, and again substituting f we get for | Odd(Result, NextVal) | (U).

    U except that Result has value   (or [illegible])
        if U(NextVal) is odd (or even).

But U(NextVal) is two characters in sequence past T(One), and hence is even or odd in step; the application of the header inverse function leaves the Result value attached to One in T. Thus the left and right sides of the recurrence equation agree for this f.

To prove that a function f is the one computed by a recursive procedure call, it is necessary that f satisfy the recurrence relation; however, this condition is not sufficient. Furthermore, when there are multiple recursions the situation is even more complex. Consider the case in which procedure P calls procedure Q and vice versa, and P does not call itself, but Q does call itself. Then in working out the meaning of a call on P the meaning of Q will appear, and working this out in turn will yield a recurrence relation defining Q's meaning in terms of itself and P's meaning. This illustrates the general situation: the result of analysis will be a system of m simultaneous equations in m functions, one for each procedure involved in a recursive chain of calls. Substitution may simplify this system (for example, substituting the P equation into the Q equation will eliminate P), but cannot solve it. It is evident that the case of recursion is far more complex than the case of WHILE-statements, and a result parallel to the WHILE verification theorem more difficult to obtain.

9. Summary and Conclusions

Using a subset of Pascal, a "program calculus" has been described that assigns functional meaning to programs. For programs containing no loops or recursive procedure calls, the application of this calculus is mechanical; in the more complicated cases, the meanings must be guessed and then checked. For recursive calls, further work must be done to obtain a simple checking procedure for the general case. To prove a program correct for a specification in the form of acceptable input-output pairs then becomes an elementary set-theoretic problem. The specification is a set of pairs as is the program meaning; the program is correct just in case its meaning is a subset of the specification.

References

1. Naur, P. et al., Revised report on the algorithmic language ALGOL 60, *CACM* 6 (1963), pp. 1-17.

2. Knuth, D. E., Semantics of context-free languages, *Math. Systems Theory* 2 (1968), pp.127-146.

3. van Wijngaarden, A. et al., Revised report on the algorithmic language ALGOL 68, *Acta Inf.* 5 (1975), pp. 1-236.

4. Wirth, N. and C. A. R. Hoare, A contribution to the development of ALGOL, *CACM* 9 (1966), pp. 413-432.

5. Wegner, P., The Vienna definition language, *Comp. Surveys* (1972), pp. 6-63.

6. Stoy, J. E., *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

7. Tennent, R. D. *Principles of Programming Languages*, Prentice Hall, 1981.

8. Mills, H. D., The new math of computer programming, *CACM* 18 (1975), pp. 43-48.

9. Linger, R. C., Mills, H. D., and Witt, B. I., *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.

10. Apt, K. R., Ten years of Hoare's logic: a survey--part I, *TOPLAS* 3 (1981), pp. 431-483.

11. Arden, B. W., *The Michigan Algorithmic Decoder*, University of Michigan Computing Center, Ann Arbor.

12. Shaw, C. J., JOVIAL--a programming language for real-time command systems, in *Annual Review in Automatic Programming*, Vol. 3, Pergamon Press, 1963, pp. 53-120.

13. Mills, H. D. et al., *Computer Programming*, Allyn and Bacon, to appear.

14. Wirth, N., The programming language Pascal, *Acta Inf.* 1 (1971), pp. 35-63.

15. Scott, D. S. and Strachey, C. Toward a mathematical semantics for computer languages, *Proc. Symposium on Computers and Automata* (J. Fox, ed.), Polytechnic Institute of Brooklyn, 1971.

16. Turing, A. M., On computable numbers, with an application to the entscheidungsproblem, *Proc. London Math. Society Ser. 2* 42 (1936), pp. 230-265.

17. Kleene, S. C., *Introduction to Metamathematics*, D. Van Nostrand, 1950.

18. Gerhart, S. and Yelowitz, L., Observations of fallibility in applications of modern programming methodologies, *IEEE Trans. Software Engineering* SE-2 (1976), pp. 195-207.